基于 MapReduce 的并行频繁项集挖掘算法研究 *

刘卫明，张 弛，毛伊敏†

(江西理工大学 信息工程学院, 江西 赣州 341099)

摘 要：针对并行 MRPrePost (parallel PrePost algorithm based on MapReduce)频繁项集挖掘算法在大数据环境存在运行时间长，内存占用量大和节点负载不均衡的问题。提出一种基于 DiffNodeset 的并行频繁项集挖掘算法—PFIMD(parallel frequent itemsets mining using DiffNodeset)。该算法首先采用一种数据结构 DiffNodeset，有效的避免了 N-list 基数过大的问题；此外提出一种双向比较策略“T-wcs”(2-way comparison strategy)，以减少两个 DiffNodeset 在连接过程中的无效计算，极大的降低了算法时间复杂度；最后考虑到集群负载对并行算法效率的影响，进一步提出了一种基于动态分组的负载均衡策略“LBSBDG”(load balancing strategy based on dynamic grouping)，该策略通过将频繁 1 项集 F-list 中的每项进行均匀分组，降低了集群中每个计算节点上 PPC-Tree 树的规模，进而减少了先序后序遍历 PPC-Tree 树所需的时间。实验结果表明，该算法在大数据环境下进行频繁项集挖掘具有较好的效果。

关键词：DiffNodeset 数据结构；MapReduce 编程模型；T-wcs 策略；LBSBDG 策略；频繁项集挖掘

中图分类号：TP311 doi: 10.19734/j.issn.1001-3695.2020.02.0039

Research on parallel frequent itemset mining algorithm based on MapReduce

Liu Weiming, Zhang Chi, Mao Yimin†

(*School of Information Engineering Jiangxi University of Science & Technology*, *Ganzhou Jiangxi* 341099, *China*)

Abstract: Aiming at the problem of excessive time, space complexity and unbalanced load for each node based on the parallel frequent itemset mining algorithm MRPrePost, this paper proposed an optimization parallel frequent itemset mining algorithm based on MapReduce, named PFIMD. Firstly, this algorithm adopted a data structure called DiffNodeset, which effectively avoid the defect that the N-list cardinality got very large in the MRPrePost algorithm. Secondly, in order to reduce the time complexity of this algorithm, it designed the T-wcs (2-way Comparison Strategy) to avoid the invalid calculation in the procession of two DiffNodesets connection. Finally, considering the impact of cluster load on the efficiency of parallel algorithm, it proposed the LBSBDG (Load Balancing Strategy Based on Dynamic Grouping) , which decreased the size of PPC-Tree on each computing node and reduced the amount of time required to traverse the PPC-Tree by evenly grouping each item in the F-list. The experimental results show that the modified algorithm has better performance on mining frequent itemset in a big data environment.

Key words: diffnodeset structure; MapReduce; 2-way comparison strategy; load balancing strategy based on dynamic grouping; frequent item mining

0 引言

随着互联网信息技术的快速发展，大数据在社交网络、电子商务、精准营销等领域得到了广泛的应用。相较于传统数据，大数据的 4V 特性：Volume(数量大)、Variety(种类多)、Velocity(速度快)、Value(价值密度低)，使得在大数据环境下的数据挖掘算法需要满足以下几个条件：a) 在大数据存储时，不仅需要存储结构化数据而且还要存储半结构和非结构化数据；b) 数据量增大的同时，数据的价值密度降低，要求算法的挖掘精度能更高；c) 新产生的数据必须要尽快处理，要求算法的实时性高[1]。鉴于此，大数据环境下的数据挖掘成为目前研究领域的一个重要主题。

数据挖掘又被称为知识发现 KDD(knowledge discover in database)，其目的在于发现大量数据中有用的信息。常见的数据挖掘任务有关联规则挖掘、分类、聚类等。其中关联规则挖掘是其重要分支之一，通过关联规则的研究能够准确的找出有用的规则，这些规则对于企业管理上的决策具有巨大帮助[2]。传统关联规则挖掘算法根据其所挖掘的数据源呈现形式分为两类：a)基于水平型数据的算法，所谓水平型是指事务记录在数据库中按行存储，典型算法有 Apriori 算法[3]和 FP-Growth 算法[4]；b)基于垂直型数据的算法，所谓垂直型是指事务记录在数据库中按列存储，代表算法有 Eclat 算法[5]。随着信息技术高速发展，大数据环境下需要处理的数据量不断增加，运行时间和内存使用量成为传统关联规则挖

掘算法的重要瓶颈，单纯的通过提升计算机硬件水平来满足人们对大数据分析与处理的需求，显得尤为困难。此时并行化的计算思想变得非常重要，通过改进传统的关联规则挖掘算法，并与分布式计算模型相结合成为当前研究的主要方向。

近年来在大数据处理方面，Google 开发的 MapReduce 并行编程模型由于其操作简单、自动容错、负载均衡、扩展性强等优点深受广大学者和企业的青睐[6]。同时 Apache 研发的 Hadoop[7]作为一种广泛使用的 MapReduce 开源框架,不仅实现了对 MapReduce 的动态调用而且在很大程度上促进了 MapReduce 的应用开发。目前许多基于 MapReduce 计算模型的关联规则挖掘算法已成功应用到大数据的分析与处理领域中。文献[8~10]采用传统 Apriori 算法多次迭代的思想，在每次迭代过程使用一个 MapReduce 任务，实现了 Apriori 算法

收稿日期：2020-02-16；**修回日期：**2020-04-22 **基金项目：**国家自然科学基金资助项目(41562019)；国家重点研发计划资助项目(2018YFC1504705)

作者简介：刘卫明(1964-)，男，江西新余人，教授，硕士，主要研究方向为矿业信息系统；张弛(1994-)，男，河南洛阳人，硕士研究生，主要研究方向为数据挖掘，并行计算等；毛伊敏(1970-)，女(通信作者)，教授，博士，主要研究方向为数据挖掘、地理信息系统等.(mymlyc@163.com).

的并行化。但是在迭代过程中需要频繁访问数据集，对数据集进行多次迭代，消耗大量的时间和空间。鉴于并行 Apriori 算法的固有缺陷，文献[11~14]中通过将 FP-Growth 算法向 MapReduce 计算模型进行迁移提出了并行 FP-Growth 算法。与并行 Apriori 算法不同，此类算法在挖掘过程中不产生候选项集，而是通过两次扫描事务数据集，在多个并行的节点上生成局部 FP-Tree 树，并对局部 FP-Tree 树进行遍历得到局部频繁项集，然后将其合并得到全局频繁项集。此外在挖掘局部频繁项集的过程中，各计算节点之间可以独立计算，既不需要相互等待也不需要相互交换数据，极大的提高了并行关联规则挖掘算法的效率。然而并行 FP-Growth 算法在挖掘过程中需要花费大量时间来递归构建频繁项 FP-Tree 树，且大数据环境下并行 FP-Growth 算法所构造的 FP-Tree 树的规模十分巨大，对于 FP-Tree 树的存储需要消耗大量的内存。考虑到并行水平格式算法的不足，文献[15~18]提出了并行的 Eclat 算法，此类算法在一定程度上克服了从海量数据集中挖掘频繁项集时存在内存和计算能力不足的问题。但并行的 Eclat 算法需要将水平数据集转换为垂直数据集作为输入数据，这对于大数据来说是无法实现的。

为减少并行计算中单个节点的内存需求量与节点之间的通信量，Liao 等人[19]结合不同算法的优势提出了一种将 dist-Eclat[15]与传统的 FP-Growth 算法[4]相结合的混合算法--MRPrePost 算法。该算法分为三个阶段：首先通过分布式计算得到频繁 1 项集 F-list；其次构造 F-list 所对应的 PPC-Tree 树，并通过对 PPC-Tree 树进行先序和后序遍历产生频繁项的 N-list；最后对 N-list 进行分组，并分布在多个节点上进行频繁项集的挖掘。相较于其他单一的并行关联规则挖掘算法，MRPrePost 算法结合了并行 FP-Growth 算法和并行垂直算法的优点，既能对原始事务集进行无损压缩又可以快速计算项集的支持度，此外该算法将对树的挖掘过程转变成与垂直格式交运算类似的 N-list 合并过程，该过程不需要将 PPC-Tree 树保存在内存中，因此极大的节省了算法的计算时间和内存空间。但是该算法仍存在几个明显不足：1)在某些数据集上频繁项的 N-list 基数过大，极易造成内存溢出；2)在合并两个频繁项的 N-list 时需要逐一比较两者中的每一项，时间复杂度较高；3)在并行挖掘频繁项集的过程中未能充分考虑到集群负载均衡对算法性能的影响。针对上述问题，本文提出了一种基于 DiffNodeset[21,22]结构的并行频繁项集挖掘算法—PFIMD(parallel frequent itemsets mining using DiffNodeset)。该算法采用了一种数据结构 DiffNodeset，有效的避免了 N-list 基数过大的问题；此外提出了一种双向比较策略“T-wcs”(2-way comparison strategy)，来减少两个 DiffNodeset 在连接过程中的无效计算，极大的降低了算法时间复杂度；最后考虑到集群负载对并行算法效率的影响，进一步提出了一种基于动态分组的负载均衡策略“LBSBDG”(load balancing strategy based on dynamic grouping)，该策略通过将频繁 1 项集 F-list 中的每项进行均匀分组，降低了集群中每个计算节点上 PPC-Tree 树的规模，进而减少了先序和后序遍历 PPC-Tree 树所需的时间。实验结果表明，该算法在大数据环境下进行频繁项集挖掘具有较好的效果。

1　算法及相关概念介绍

1.1　PrePost 算法相关定义

定义 1　PPC-Tree[20]。PPC-Tree 树是一种树型数据结构，树中的每个节点均由以下五部分组成：

a) Item-name:节点名称

b) count:节点计数

c) pre-order:先序遍历序号

d) post-order:后序遍历序号

e) children-list:孩子节点集合

定义 2　PP-code[20]。PP-code 编码又被称为先序后序编码，主要由 pre-order、post-order 和 count 三部分组成。对于 PPC-Tree 树中的任意节点 N，则称 $(N.\text{pre-order}, N.\text{post-order}, N.\text{count})$ 为该节点的 PP-code 编码。

性质 1　祖先孩子关系[20]。给定 PPC-Tree 中任意两节点 N_1 和 N_2 ($N_1 \neq N_2$)的 PP-code，若满足 $N_1.\text{pre-order} < N_2.\text{pre-order}$，$N_1.\text{post-order} > N_2.\text{post-order}$ 则称节点 N_1 是节点 N_2 的祖先节点，N_2 为 N_1 的孩子节点。

定义 3　频繁 1 项集的 N-list[20]。在 PPC-Tree 树中，代表相同项的所有 PP-code 编码根据先序遍历顺序连接生成的序列，被称为频繁 1 项集的 N-list。

性质 2　频繁 1 项集的支持度[20]。给定项 item，其 N-list 为 $(x_1,y_1,z_1),(x_2,y_2,z_2),...,(x_m,y_m,z_m)$，则项 item 的支持度为 $z_1+z_2+,...,+z_m$。

1.2　DiffNodeset 相关定义

定义 4　‘$\prec$’关系[21,22]。给定频繁 1 项集中的任意两项 i_1 和 i_2，若 i_2 的支持度大于 i_1 的支持度，则表示为 $i_1 \prec i_2$。

定义 5　2 项集的 DiffNodeset[21,22]。给定频繁 1 项集中的任意两项 i_1 和 i_2 ($i_1 \prec i_2$)，其 N-list 结构分别为 N-list_1，N-list_2。2 项集 i_1i_2 的 DiffNodeset 结构记为 $DiffNodeset_{12}$，定义如下：

$$DiffNodeset_{12} = \{(x.\text{pre-order}, x.\text{post-order}, x.\text{count}) \mid x \in \text{N-list}_1 \wedge \neg(y \in \text{N-list}_2)\} \quad (1)$$

其中 y 对应节点是 x 对应节点的祖先节点。

性质 3　2 项集的支持度[21,22]。给定 2 项集 i_1i_2，其 DiffNodeset 表示为 $(x_1,y_1,z_1),(x_2,y_2,z_2),...,(x_n,y_n,z_n)$，若 i_1 的支持度为 $\text{support}(i_1)$，则 i_1i_2 的支持度等于 $\text{support}(i_1)-(z_1+z_2+,...,+z_n)$。

定义 6　k 项集的 DiffNodeset[21,22]。假设 k 项集 $P=i_1i_2...i_{k-1}i_k$，$(i_1 \prec i_2 \prec ... \prec i_{k-1} \prec i_k)$，频繁 $k-1$ 项集 $P_1=i_1i_2...i_{k-2}i_{k-1}$，$P_2=i_1i_2...i_{k-2}i_k$ 所对应的 DiffNodeset 分别记为 DN_1，DN_2。项集 P 的 DiffNodeset 记为 DN_P,其计算公式如下：

$$DN_P = DN_2 / DN_1 \quad (2)$$

其中‘/’表示集合差。

性质 4　k 项集的支持度[21,22]。给定 k 项集 $P=i_1i_2...i_{k-1}i_k$，其 DiffNodeset 结构表示为 $(x_1,y_1,z_1),(x_2,y_2,z_2),...,(x_l,y_l,z_l)$，若项集 $P_1=i_1i_2,...,i_{k-2}i_{k-1}$ 的支持度为 $\text{support}(P_1)$，则项集 P 的支持度等于 $\text{support}(P_1)-(z_1+z_2+...+z_l)$。

2　PFIMD 算法

PFIMD 算法主要包括三个阶段：挖掘频繁 1 项集、频繁 1 项集均匀分组和并行挖掘频繁项集。在第一阶段主要通过调用一次 MapReduce 任务来并行获取频繁 1 项集 F-list；在第二阶段中考虑到集群负载均衡对并行算法挖掘效率的影响，使用动态分组策略 LBSBDG 将 F-list 中的每一项进行均匀分组，从而生成哈希表 G-list；在第三阶段中主要包含并行挖掘频繁项集的 Map 阶段和 Reduce 阶段，其中在 Map 阶段各个节点根据前两个阶段产生的 F-list 和 G-list 来构造映射路径，在 Reduce 阶段各节点首先根据映射路径构造子 PPC-Tree 树，然后根据子 PPC-Tree 树来挖掘局部频繁项集，最后合并得到全局频繁项集。

2.1　挖掘频繁 1 项集

对于数据集 DB，其频繁 1 项集的生成过程主要包括 Split、Map、Combine 以及 Reduce 四个阶段。a)在 Split 阶段：使用 Hadoop 默认的文件块策略，将原始数据集划分成大小相同的文件块 Block；b)在 Map 阶段：文件块 Block 作为输入数据，Mapper 节点通过调用 Map 函数以键值对 $<key=\text{item}, value=1>$ 的形式统计出相应节点中各项出现的次数；同时为了降低集

群中各节点的数据通信量，经过 Mapper 节点处理后的数据不会立刻发送给 Reducer 节点，而是先通过 combiner 进行本地结合；c)在 Combine 阶段：本地节点中 key 值相同的键值对进行累加，经过本地合并处理后的键值对会被自动分配到不同的 Reducer 节点，同时 key 值相同的键值对分配到相同的 Reducer 节点；d)在 Reduce 阶段：需要对这些键值对的 value 值进行再次合并，即可得到每个数据项 key 在整个数据集中的支持数，最后根据最小支持度 min_sup 筛选出频繁 1 项集 F-list。

2.2　频繁 1 项集均匀分组

针对大数据环境下频繁 1 项集 F-list 规模太大，无法在有限的内存空间中构造 PPC-Tree 树的问题。本文提出了一种动态分组策略“LBSBDG”，该策略通过对 F-list 进行均匀分组，将原事务数据集各记录中的频繁项根据分组结果进行重新划分，并将原 PPC-Tree 树分割成多个独立的子树，使得每个节点上的子 PPC-Tree 树能够适应内存大小。采用“LBSBDG”分组策略对频繁 1 项集 F-list 进行均匀分组时，其关键在于计算 F-list 中每一项的负载量，即频繁 1 项集中每个项所对应 N-list 结构的长度。然而 N-list 中的元素与 PPC-Tree 树中的节点一一对应，在未构造 PPC-Tree 树之前无法准确计算出每一项的负载量。为了解决该问题，在“LBSBDG”分组策略中通过估计函数 E(item) 对频繁 1 项集 item 的长度规模进行预测。

定义 7　负载量估计函数 E(item) 。若频繁项 item 的支持度为 count，且在 F-list 中的位置为 l，则其负载量估计函数如下所示。

$$E(\text{item}) = \min\{\text{count}, 2^{l-1}\} \tag{3}$$

证明　对于频繁项 item 来说，其 N-list 的长度表示该项在 PPC-Tree 树中的节点个数，显然对于每一项来说节点数的最大值为该项的支持度。而且在构造 PPC-Tree 树时，树中每一项的节点数与其自身在 F-list 序列中的位置有关。对于频繁 1 项集 item 来说，假设其在 F-list 的位置为 l，则最坏情况是排在 item 之前的 $l-1$ 项中任意项组合在 PPC-Tree 中都有对应的路径，且该路径也包含项 item，在此情况下这样的路径最多有 2^{l-1} 条。因此 F-list 中的每一项 item 的 N-list 长度不超过 2^{l-1} 与该项支持度之间的较小值。

对于频繁 1 项集 F-list，采用“LBSBDG”分组策略将其划分为 G 组的分组思想如下：首先根据式(3)计算出 F-list 中每一项的负载量，并根据负载量的降序排列生成 L-list，选取 L-list 中的前 G 项作为初值依次添加到 0~(G-1)组，并更新每一组的负载总量；然后继续对 L-list 中未分组的项进行分组操作，每次读取 G 项，在划分之前需要判定当前每组的负载总量是否相同，若每组的负载总量均相同则顺序添加，即将 G 项分别添加到 0~(G-1)组，若不相同则逆序添加，即将 G 项分别添加到(G-1)~0 组中，如果 L-list 中未分组项的个数小于 G 则将其依次添加到负载量最小的组中；最后将所得到的分组结果 G-list 保存到分布式文件系统 HDFS 中，从而使得集群中任意节点都能访问到 G-list。

“LBSBDG”分组过程的形式化如算法 1 所示。

算法 1　“LBSBDG”分组策略

```
输入：频繁 1 项集 F-list，分组数 G
输出：分组结果 G-list
a) 计算 F-list 中每一项的负载量
   L-list ← ∅
Foreach item in F-list do
Compute item_load by Eq.(3)
   L-list ← <key = item,value = item_load>
Sorted(L-list) //L-list 按 value 值进行非递减排序
End for
b) F-list 均匀分组
   G-list ← {∅_1,∅_2,...,∅_G}
If L-list.length >= G do
insert_item ← {item_0,item_1,...,item_{G-1}} //每次选取 L-list 中的前 G 项
If 每组的负载量均相同 do
G-list[i] ← item_i //按顺序将每项添加到 0~(G-1)组
G-list[i].load += item_i.load //更新每组负载量
Else do //如果每组负载量不全相同
G-list[G − i − 1] ← item_i //逆序将每项添加到(G-1)~0 组
   G-list[G - i - 1].load += item_i.load
Del( insert_item )//删除已添加项
End if
Else do //如果 L-list 中剩余不足 G 项
Foreach item in L-list do
gid = getMinload(G-list) //获取目前负载量最小的组号
   G-list[gid] ← item
   G-list[gid].load += item.load
End for
End if
Output(G-list)
```

2.3　并行挖掘频繁项集

采用“LBSBDG”分组策略对 F-list 进行均匀分组是为了将原始事务数据集中的事务进行重新划分，并把划分后的事务集映射到集群中的各个计算节点上。通过在各个节点上构建子 PPC-Tree 树，来完成频繁项集的挖掘任务。在此过程中主要包括并行挖掘频繁项集的 Map 阶段和 Reduce 阶段，同时为了降低内存消耗，需要对原始数据集进行预处理，用项 item 在 F-list 中的位置来替换原始数据集中的 item。经过预处理后，各个节点启动新的 MapReduce 任务进行频繁项集的挖掘。

2.3.1 Map 阶段

在并行挖掘频繁项集的 Map 阶段，其主要任务是将预处理后的数据跟据哈希表 HTable 映射到集群中不同的计算节点上，其具体过程为：首先从分布式文件系统 HDFS 中读取全局频繁 1 项集 F-list 和算法 1 中所获得的哈希表 G-list，并将 G-list 每组所包含的项作为 key，组号 gid 作为 value 构建 HTable；之后依次读取预处理后的每一条记录，逆序遍历该记录中的项 item，根据之前得到的 HTable，确定其组号 gid，然后以 gid 为 key，排在 item 之前所有项为 value 组成键值对。与此同时为了避免同一条记录多次映射到同一节点，删除 HTable 中 value= gid 的所有键值对。如果在映射时找不到对应的组号，则读取前一项执行相同操作，直到该记录执行完毕；最后将所得的所有结果作为 Reduce 阶段的输入传送给 Reduce 函数。其操作过程如算法 2 所示。

算法 2　并行挖掘频繁项集的 Map 过程

```
输入：预处理后的数据 pre_data，F-list，G-list
输出：映射路径 <key, value=path>
a) 建立哈希表 HTable
从 HDFS 中读入 G-list
Foreach g in G-list do
Foreach item in g do
  Htable[g[item]]=gid
End for
End for
b) 生成映射路径
Foreach trans in pre_data do
items[]←Split(trans) //将输入的数据进行分解并保存到空
```

```
数组 items[]中
3 For j=len(items)-1 to 0 do
4 If Htable[items[j]] not null do //判断 items[j]所在路径属于哪个组
path={items[0],items[1],…,items[j]}
5 End if
output(key=Htable[items[j]],value=path)
del(Htable[items[j]])
6 End for
7 End for
```

2.3.2 Reduce 阶段

在 Reduce 阶段，首先采用 DiffNodeset 数据结构避免 N-list 基数过大的问题。此外在合并两个频繁 1 项集的 N-list 过程中使用双向比较策略“T-wcs”能够极大减少比较次数，从而加快完成频繁 1 项集的 N-list 合并任务。

性质 5 序列一致性原则。对于频繁项 N，其 N-list 表示为 $\{(x_1,y_1,z_1),(x_2,y_2,z_2),\ldots,(x_n,y_n,z_n)\}$，则有 $x_1<x_2<\ldots<x_n$，$y_1<y_2<\ldots<y_n$。

证明 跟据 N-list 的定义本文可知 $x_1<x_2<\ldots<x_n$。假设 (x_1,y_1,z_1) 对应节点 N_1，(x_2,y_2,z_2) 对应节点 N_2，由于 N_1.item-name $=N_2$.item-name，则 N_1 与 N_2 不存在祖先孩子关系，而 $x_1<x_2$ 说明 N_2 相较于 N_1 来说一定在右子树上，跟据后序遍历的规则，一定存在 N_1 的后序遍历序列小于 N_2 的后序遍历序列，即 $y_1<y_2$，依此类推 $y_1<y_2<\ldots<y_n$ 成立。

在并行挖掘频繁项集过程中最重要的一步是将频繁 1 项集的 N-list 结构合并产生 2 项集的 DiffNodeset 结构，如何降低该过程的运行时间是该算法的关键所在。为此本文提出了一种双向搜索策略“T-wcs”，该策略通过利用序列一致性原理和祖先孩子关系能够大大减少合并过程中所需要比对的次数。

给定两个频繁 1 项集 i_1,i_2 $(i_1,i_2\in$ F-list $\wedge\, i_1\prec i_2)$，其 N-list 分别表示为 N-list$_{i1}$ 和 N-list$_{i2}$，且长度为 m 和 n，其具体形式如下：

$$\text{N-list}_{i1}=\{(x_{11},y_{11},z_{11}),(x_{12},y_{12},z_{12}),\ldots,(x_{1m},y_{1m},z_{1m})\} \quad (4)$$

$$\text{N-list}_{i2}=\{(x_{21},y_{21},z_{21}),(x_{22},y_{22},z_{22}),\ldots,(x_{2n},y_{2n},z_{2n})\} \quad (5)$$

在比较两者中的任意项 (x_{1a},y_{1a},z_{1a}) 和 (x_{2b},y_{2b},z_{2b}) 时，根据序列一致性原则只存在三种情况：

a) $y_{1a}>y_{2b}$，$(1\le a\le m,1\le b\le n)$。根据祖先孩子关系可知 (x_{2b},y_{2b},z_{2b}) 在 PPC-Tree 树中所对应的节点 $N_{i2}[b]$ 不是 (x_{1a},y_{1a},z_{1a}) 所对应节点 $N_{i1}[a]$ 的祖先节点。此外根据序列一致性原理可知频繁 1 项集的 N-list 是按照 post-order 的升序序列排序，所以 $N_{i2}[b]$ 也不是 N-list$_{i1}$ 中排在 $N_{i1}[a]$ 之后元素所对应的祖先节点，选择 $N_{i2}[b]$ 下一个节点进行比较。

b) $x_{1a}>x_{2b},y_{1a}<y_{2b}$，$(1\le a\le m,1\le b\le n)$。根据祖先孩子关系可知 (x_{2b},y_{2b},z_{2b}) 在 PPC-Tree 树中所对应的节点 $N_{i2}[b]$ 是 (x_{1a},y_{1a},z_{1a}) 所对应节点 $N_{i1}[a]$ 的祖先节点。所以节点 $N_{i1}[a]$ 不包含在 2 项集 i_1i_2 的 DiffNodeset 中，选择 $N_{i1}[a]$ 的下一个元素进行比较。

c) $x_{1a}<x_{2b},y_{1a}<y_{2b}$，$(1\le a\le m,1\le b\le n)$。根据祖先孩子关系可知 (x_{2b},y_{2b},z_{2b}) 所对应的节点 $N_{i2}[b]$ 不是 (x_{1a},y_{1a},z_{1a}) 所对应节点 $N_{i1}[a]$ 的祖先节点，此外 $N_{i1}[a]$ 也不可能是 $N_{i2}[j]$ 之后任意元素的孩子节点，满足 2 项集 DiffNodeset 定义。故而将 $N_{i1}[i]$ 插入到 2 项集 i_1i_2 的 DiffNodeset 中，并选择 $N_{i1}[i]$ 的下一个节点进行比较。

其操作过程如算法 3 所示。

算法 3 双向搜索策略 T-wcs 的执行过程

```
输入：频繁 1 项集的 N-list 结构
输出：2 项集的 DiffNodeset
Procedure T-wcs (i1i2)
DN_i1i2 ← ∅
1 a←0，b←0
2 l1← N-list_i1 的长度，l2← N-list_i2 的长度
3 while a<l1 ∧ b<l2 do
4 If N_i1[a].post-order > N_i2[b].post-order do
b←b+1
5 Else
6 If N_i1[a].post-order < N_i2[b].post-order ∧
7 N_i1[a].pre-order > N_i2[b].pre-order do
a←a+1
8 Else
DN_i1i2 ← DN_i1i2 ∪ {N_i1[a]}
a←a+1
9 End if
10 End if
11 End while
12 If a<l1 then
13 while a<l1 do
DN_i1i2 ← DN_i1i2 ∪ {N_i1[a]}
a←a+1
14 End while
15 End
16 Output DN_i1i2
```

在并行挖掘频繁项集的 Reduce 阶段，系统中每个计算节点先要根据 Map 阶段输出的键值对，通过调用 insert_tree() 函数在各个节点中构造 PPC-Tree 树，并对 PPC-Tree 树进行先序、后序遍历，从而得到频繁 1 项集的 N-list，同时为了减少内存消耗，通常将 PPC-Tree 树从内存中删除；然后采用双向搜索策略“T-wcs”和性质 3 将频繁 1 项集的 N-list 结构进行合并得到频繁 2 项集的 DiffNodeset；最后根据定义 6 和性质 4 迭代挖掘出所有的频繁项集。

算法 4 并行挖掘频繁项集的 Reduce 过程

```
输入：最小支持度 min_sup,
    映射路径<key=Htable[Items[j]], value=path>
输出：频繁项集 F_item
a) 构造 PPC-Tree
F_item ← ∅
1 root ← null //PPC-Tree 初始化为空
2 Foreach t in value do
3 curNode ← root //插入当前节点
4 Foreach item in t do
5 Call insert_tree(curNode,item)
  //调用 insert_tree()函数构造 PPC-Tree 树
6 End for
7 End for
F_item=F_item U F1
b) 生成频繁 1 项集的 N-list
1 scan_by_pre(root) //先序遍历 PPC-Tree
2 scan_by_post(root) //后序遍历 PPC-Tree
3 NL1[] ← N-list_Constructtion(root)
  //生成所有频繁 1 项集的 N-list
c) 生成频繁 2 项集的 DiffNodeset
F2 ← ∅
F1 ← items
1 Foreach item in F1 do
2 If groupID(item)==key do
3 DN12 ← T-wcs(item)//调用 T-wcs 生成以 item 开头的 DiffNodeset
4 Foreach item in DN12 do
5 Compute item_sup by property(3)
6 If item_sup ≥min_sup do
```

```
F2.add(item)
7 End if
8 End for
9 End if
10 End for
F_item=F_item ∪ F2
d) 递归生成所有频繁项集
1 Function: Mining_Fre_Items(DN_k, min_sup)
2 Foreach DN_k[i], DN_k[j] in DN_k do
3 DN1 ← DN_k[i].DiffNodeset ,
  DN2 ← DN_k[j].DiffNodeset
  DN12 = DN1 ∪ DN2
  DN12.DiffNodeset = DN1 / DN2
4 Calculate sup(DN12) by property(4)
5 If sup(DN12) ≥ min_sup do
  DN_k_1.add(DN12,DN12.DiffNodeset)
6 End if
7 End for
  F_item ← F_item ∪ DN_k_1
8 If DN_k_1 not null do
9 Mining_Fre_Items(DN_k_1, min_sup) //递归调用频繁项集
10 End if
11 Output ( F_item )
```

2.4 PFIMD 算法步骤

PFIMD 算法的具体实现步骤如下所示。

a) 通过 Hadoop 默认的文件块策略，将原始数据集划分成大小相同的文件块 Block;

b) 对于每一个文件块调用频繁 1 项集 F-list 的生成过程，获得全局频繁 1 项集，并将结果存入分布式缓存系统 HDFS 中;

c) 调用均匀分组策略 LBSBDG 将 F-list 中的每一项进行分组，生成分组结果 G-list;

d) 启动新的 MapReduce 任务，在 Map 阶段调用算法 2 将原始数据集分别映射到各个计算节点上，并在 Reduce 阶段调用算法 4 迭代挖掘全局频繁项集 F_item。

2.5 算法分析

PFIMD 算法主要包括三个阶段：挖掘频繁 1 项集、频繁 1 项集均匀分组和并行挖掘频繁项集，因此该算法的时间复杂度主要有三部分组成。在挖掘频繁 1 项集阶段的 Map 阶段需要遍历事务数据集每一条记录中的每一项，假设每个 Mapper 节点上的记录数目为T_D，记录的平均长度为L，则其时间复杂度为$O(T_D*L)$；在 Reduce 阶段需要将每个计算节点上的 key 值相同的键值对进行合并，假设每个 Reducer 节点中项的平均个数为T_{item}，Mapper 节点个数为M，则其时间复杂度为$O(T_{item}*M)$，因此获取频繁 1 项集的总时间复杂度为$O(T_D*L+T_{item}*M)$。在频繁 1 项集均匀分组阶段主要是采用 LBSBDG 将中的每一项进行分组，假设其 F-list 长度为l，分组数为G，则其时间复杂度为$O(l/G)$，即为$O(l)$。在并行挖掘频繁项集过程中只需将当前频繁项集和频繁项集的 DiffNodeset 结构保存在内存中，极大的降低了内存的开销。同时该过程的时间复杂度主要取决于将频繁 1 项集的 N-list 结构合并生成 2 项集的 DiffNodeset 结构，假设频繁 1 项集$\text{F-list}=\{I_1,I_2,...,I_l\}$，项$I_i$所对应的 N-list 长度为$L_i$，采用搜索策略 T-wcs 生成 2 项集 DiffNodeset 时间复杂度为$O(\sum\sum(L_a+L_b))$，其中 a，b 的取值范围为(1,l)，因此对于 PFIMD 算法来说其时间复杂度为$O(T_D*L+T_{item}*M+l+\sum\sum(L_a+L_b))$。在 MRPrePost 算法中前两个阶段的时间复杂度基本相同，而在合并两个频繁 1 项集 N-list 过程中需要逐一比较两个 N-list 结构之间的每个元素其时间复杂度为$O(\sum\sum(L_a*L_b))$。因此可知 PFIMD 算法的时间复杂度更小。

3 实验结果及比较

3.1 实验环境

为了验证 PFIMD 算法的挖掘性能，本文设计了相关实验。实验环境包含 4 节点，其中 1 个 Master 节点，3 个 Slaver 节点。所有节点的 CPU 为 AMD Ryzen 7，每个 CPU 都拥有 8 个处理单元，其内存均为 16G，且四个节点处于同一个局域网，并通过 200Mb/s 以太网相连。在软件方面每个节点安装的 Hadoop 版本为 2.7.4，java 版本为 JDK 1.8.0，操作系统均为 Ubuntu 16.04。各个节点的具体配置情况如表 1 所

表 1 实验环境中各节点的基本配置

Tab. 1 The foundation configuration of each node in the experimental environment

节点类型	主机名	IP 地址	角色
master	master	192.168.1.109	master/JobTracker/NameNode
slaver	slaver_1	192.168.1.110	slaver/TaskTracker/DateNode
slaver	slaver_2	192.168.1.111	slaver/TaskTracker/DateNode
slaver	Slaver_3	192.168.1.112	Slaver/TaskTracker/DateNode

3.2 实验数据

PFIMD 算法所使用的实验数据为三个真实的数据集，分别为 Susy、webdocs 和 kosarak。Susy[23]是一个记录使用粒子加速器探测粒子的数据集，该数据集包含 5000000 条实例，共有 190 项，具有数据量多，记录长度均匀，项数少等特点；webdocs 数据集是由意大利科学家 Claudio Lucchese 等人通过网络爬虫抓取的 Web 文档数据,该数据集包含 1692082 条数据，共有 5267656 个项，具有数据量多，记录长度长，项数多等特点[24]；kosarak 数据集[23]记录了匈牙利一家大型新闻网站点击流数据,该数据集包含 990002 条数据,共有 41270 项，具有数据量少，项数较多，数据离散等特点。数据集的具体信息如表 2 所示。

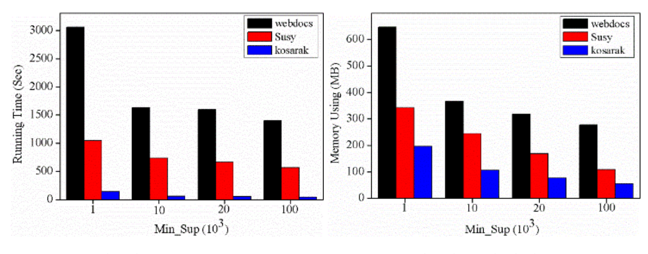
表 2 实验数据集

Tab. 2 Datasets used in the experimental

	Susy	webdocs	kosarak
记录数(条)	5000000	1692082	990002
项数(个)	190	5267656	41270
大小(MB)	321	1481.9	32.1

3.3 PFIMD 算法的性能分析

为验证 PFIMD 算法在大数据环境下挖掘频繁项集的可行性，本文选取最小支持度阈值为 1000，10000，20000 和 100000，分别将 PFIMD 算法应用于 Susy、webdocs 和 kosarak 三个数据集中独立运行 10 次，取 10 次结果的均值，通过对算法运行时间和内存使用量的比较，从而实现对 PFIMD 算法性能的综合评估。图 1 为 PFIMD 算法在 3 个数据集的执行结果。从图 1 中可以看出当支持度从 1000 变化到 10000 时，该算法在三种数据集的运行时间和内存使用量都有较大的下降。这是因为随着支持度的增大，频繁 1 项集 F-list 的规模急剧下降，采用“LBSBDG”策略分配到每个计算节点上的项也有所减少，同时也降低了每个计算节点中子 PPC-Tree 树的规模，极大的减少了产生 N-list 结构所需的时间。此外，使用“T-wcs”搜索策略生成 2 项集的 DiffNodeset 时的时间复杂度是线性的，且在挖掘频繁项集时只需要将以当前项为前缀的频繁项集保存在内存中，极大的降低了内存占用量。然而随着支持度持续增加，算法的运行时间和内存使用量减小的趋势越来越缓慢，这是因为 MapReduce 计算模型工作调度以及在中间结果的 I/O 上占用了大部分时间从而影响了算法的性能。



(a)算法的运行时间　　(b)算法的内存使用量

图 1　PFIMD 算法的性能分析

Fig. 1　The performance of PFIMD

3.4　PFIMD 算法性能比较

3.4.1 PFIMD 算法时间复杂度比较

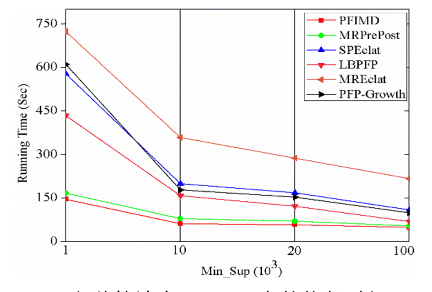
为验证 PFIMD 算法的挖掘效果，本文将 PFIMD 算法分别与 PFP-Growth 算法[11]，LBPFP 算法[14]，MREclat 算法[17]，SPEclat 算法[18]和 MRPrePost 算法[19]进行了对比。在执行上述并行算法时需要根据每个数据集的 F-list 规模设置分组数目，表 3 给出三种数据集在不同支持数下 F-list 数目的具体情况。根据 F-list 大小对于 Susy 数据集设置分组数为 50 组，kosarak 数据集设置分组数为 100 组，webdocs 数据集设置分组数为 1000 组，分别在上述三个数据集上运行六种并行算法，实验结果如图 2 所示。

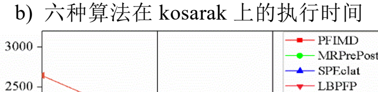
表 3　不同支持度下各数据集的 F-list 规模

Tab. 3　F-list size of each dataset under different support degrees

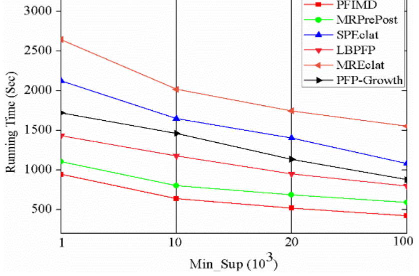
数据集	1000	20000	100000
webdocs	17104	2420	556
kosarak	1253	102	32
Susy	117	99	91

a) 六种算法在 webdocs 上的执行时间





b) 六种算法在 kosarak 上的执行时间



c) 六种算法在 Susy 上的执行时间

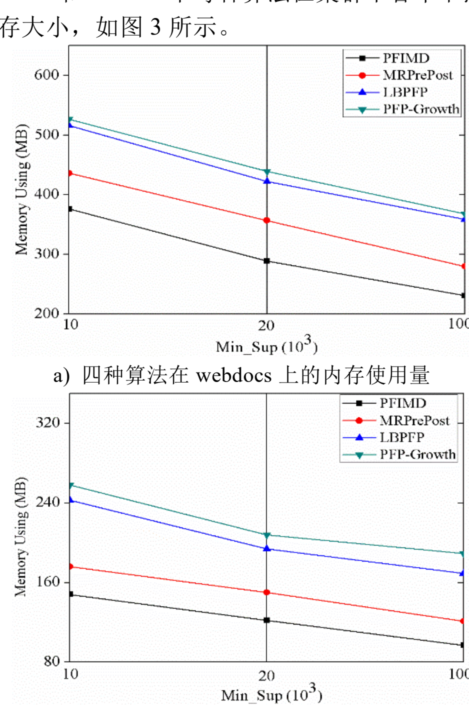
图 2　不同算法在不同数据集上的运行时间比较

Fig. 2　Comparison of time complexity of different algorithms

从图 2 中可以看出，相较于其他算法，PFIMD 算法在各个数据集上的运行时间均有降低，其中在 kosarak 降低的最多，PFIMD 算法比 SPEclat，MREclat，PFP-Growth 和 LBPFP 算法的执行时间分别降低了 66.0%，79.5%，62.1%和 52.1%；在 webdocs 数据集上降低的最少，但也分别降低了 31.5%,37.6%,17.3%和 17.06%。这是由于在并行挖掘频繁项集过程中 PFIMD 算法将对树的遍历转换为对 DiffNodeset 数据结构的合并任务，极大的降低了算法的运行时间。相反，在挖掘频繁项集时，SPEclat 算法和 MREclat 算法需要先将水平数据集转为垂直数据集，然后采用类 Apriori 算法通过多次迭代来完成频繁项集的挖掘，同样对于 LBPFP 算法和 PFP-Growth 算法需要递归的构建频繁项集的条件模式树，这两者都需要消耗大量的时间。此外，可以发现 PFIMD 算法比最优的 MRPrePost 算法的挖掘效果都好，尤其在 Susy 数据集上，PFIMD 算法的执行时间比 MRPrePost 算法降低了 21.8%。主要因为 PFIMD 算法采用双向搜索策略“T-wcs”使得生成 2 项集的 DiffNodeset 时间复杂度是线性的，并且 PFIMD 算法在并行挖掘频繁项集时采用动态分组策略“LBSBDG”，均匀的将频繁 1 项集分配到各个计算节点中，在确保集群负载均衡的同时也减小了集群中各节点中子 PPC-Tree 树的规模，因此减少了先序后序遍历子 PPC-Tree 树所需要的时间，进一步降低了 PFIMD 算法的运行时间。

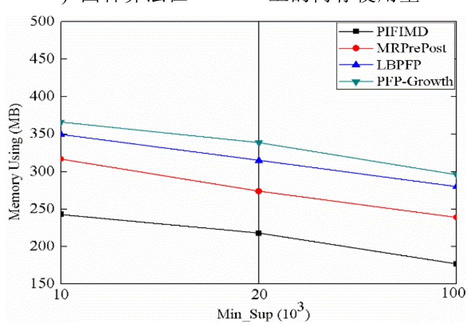
3.4.2 PFIMD 算法空间复杂度比较

由于 PFP-Growth 算法、LBPFP 算法、MRPrePost 算法和 PFIMD 算法都对原始数据集进行了无损压缩，因此本文除了考察并行算法的运行时间外，还统计了支持度为 10000,20000 和 100000 下每种算法在集群中各个节点消耗的平均内存大小，如图 3 所示。



a) 四种算法在 webdocs 上的内存使用量

b) 四种算法在 kosarak 上的内存使用量



c) 四种算法在 Susy 上的内存使用量

图 3　不同算法在不同数据集上的内存使用量比较

Fig. 3　Comparison of space complexity of different algorithms

从图 3 可以看出，在三个数据集上 MRPrePost 算法和 PFIMD 算法所消耗的内存大小明显小于 LBPFP 算法和 PFP-Growth 算法，这是由于 MRPrePost 算法和 PFIMD 算法在挖掘频繁项集时只需要根据 PPC-Tree 树生成频繁 1 项集的 N-list 结构，之后将 PPC-Tree 树从内存中删除，节省了大量的内存空间，而对于 LBPFP 算法和 PFP-Growth 算法在挖掘频繁项集时需要递归构造条件模式子树，所有的条件模式子树都需要保留在内存中。同时相较于 MRPrePost 算法，PFIMD 算法在对三个数据集挖掘频繁项集时所使用的内存空间更少，尤其在 Susy 数据集上，其内存使用量比 MRPrePost 算法减少了 22.7%。一方面是因为 PFIMD 算法使用双向搜索策略“T-wcs”，每组在挖掘时只需要将以当前项为前缀的频繁项集保存在内存中，极大的降低了内存占用量，而且采用动态分组策略“LBSBDG”，均匀的将频繁 1 项集分配到各个计算节点中减小了各节点中子 PPC-Tree 树的规模；另一方面由于 PFIMD 算法采用 DiffNodeset 结构避免了在数据集上 N-list 基数较大的问题，如表 4 所示，本文对 Susy、webdocs、kosarak 三个数据集的频繁项集的 DiffNodeset 结构和 N-list 结构基数进行了统计，从表中可以看出在各个数据集上频繁项集的 DiffNodeset 结构比 N-list 结构的规模小，尤其对于密集型数据集 Susy 来说 DiffNodeset 结构的优势更明显。

表 4　DiffNodeset 结构与 N-list 结构对比

Tab. 4　The comparison of DiffNodeset and N-list

dataset	Min_Sup	Avg.DiffNodeset	Avg.N-list	reduction ration
Susy	15%	74	972	13.1
webdocs	1%	3604	5793	1.6
kosarak	0.2%	282	2091	7.4

4　结束语

为解决传统频繁项集挖掘算法在大数据环境下的不足，本文提出了一种新的并行频繁项集挖掘算法 PFIMD。该算法首先采用 DiffNodeset 数据结构，有效避免了 PrePost 算法中 N-list 基数过大的问题；其次，提出一种双向搜索策略“T-wcs”，以加快 2 项集的 DiffNodeset 生成任务，降低算法的时间复杂度；最后，结合 Hadoop 云计算平台和 MapReduce 编程模型对改进后的 ProPost 算法各步骤进行并行化处理，并且提出了一种基于动态分组的负载均衡策略“LBSBDG”，解决了集群各节点负载不均衡的问题。与其他并行频繁项集挖掘算法相比，该算法充分结合了水平型算法和垂直型算法的优点，利用其独特优势来实现频繁项集的挖掘。同时为了验证 PFIMD 算法的挖掘性能，本文在 Susy、webdocs、kosarak 三个数据集上对 PFIMD、MRPrePost、PFP-Growth、MREclat、LBPFP 和 SPEclat 六种算法进行对比分析，实验结果表明相比于其他几种算法，PFIMD 算法在运行时间和内存使用量上都具有明显的优势。

参考文献：

[1] 米云龙，米春桥，刘文奇．海量数据挖掘过程相关技术研究发展 [J]．计算机科学与研究, 2015, 9 (6): 641-659. (Mi Yunlong, Mi Chunqiao, Liu Wenqi. Research advance on related technology of massive data mining process [J]. Journal of Frontier s of Computer Science and Technology, 2015, 9 (6): 641-659.)

[2] 肖文，胡娟，周晓峰．基于 MapReduce 计算模型的并行关联规则挖掘算法研究综述 [J]．计算机应用研究, 2018, 35 (1): 13-23. (Xiao Wen, Hu Juan, Zhou Xiaofeng. Parallel association rules mining algorithm based on MapReduce: a survey [J]. Application Research of Computers, 2018, 35 (1): 13-23.)

[3] Agrawal R, Srikant R. Fast algorithm for mining association rules [J]. Proc 20th Int Conf Very Large Data Bases (VLDB) , 1994. 23 (3): 21-30.

[4] Han Jiawei, Pei Jian, Yin Yiwen. Mining frequent patterns without candidate generation: A frequent-pattern tree approach [J]. Data Mining & Knowledge-Based Systems, 2004, 8 (1): 53-87.

[5] Zaki M, Parthasarathy S, Ogihara M. New algorithms for fast discovery of association rules [C]// Proc of the 3rd International Conference on Knowledge Discovery and Data Mining, 1999: 283-286.

[6] 黄山，王波涛，王国仁等．MapReduce 优化技术综述 [J]．计算机科学与探索, 2013, 7 (10): 865-885. (Huang Shan, Wang Baotao, Wang Guoren, *et al.* A survey on MapReduce optimization technologies [J]. Journal of Frontiers of Computer Science and Technology, 2013, 7 (10): 865-885.)

[7] Afzali M, Singh N, Kumar S. Hadoop-MapReduce: a platform for mining large datasets [C]// International Conference on Computing for Sustainable Global Development. Piscataway, NJ: IEEE Press, 2016.

[8] 黄立勤，柳燕煌．基于 MapReduce 并行的 Apriori 算法改进研究 [J]．福州大学学报：自然科学版, 2011, 39 (5): 69-74. (Huang Liqin, Liu Yanhuang. Research on improved parallel Apriori with MapReduce [J]. Journal of Fuzhou University: Natural Science Edition, 2011, 39 (5): 69-74.)

[9] Zhou Xiaohao, Huang Yongfeng. An improved parallel association rules algorithm based on MapReduce framework for big data [C]// Proc of the 10th International Conference on Natural Computation, 2014: 284-288.

[10] 秦军，郝天曙，董倩倩．基于 MapReduce 的 Apriori 算法并行化改进 [J]．计算机技术与发展, 2017, 27 (4): 64-68. (Qin Jun, Hao Tianshu, Dong Qianqian. Parallel improvement of Apriori algorithm based on MapReduce [J]. COMPUTER TECHNOLOGY AND DEVELOPMENT, 2017, 27 (4): 64-68.)

[11] Li Haoyuan, Wang Yi, Zhang Dong. PFP: parallel FP-growth for query recommendation [C]// Proc of ACM Conference on Recommender systems. New York: ACM Press, 2008: 107-114.

[12] 杨勇，王伟．一种基于 MapReduce 的并行 FP-growth 算法 [J]．重庆邮电大学学报：自然科学版, 2013, 25 (5): 651-657, 670. (Yang Yong, Wang Wei. A parallel FP-growth algorithm based on MapReduce [J]. Journal of Chongqing University of Posts and Telecommunications (Natural Science Edition) , 2013, 25 (5): 651-657, 670.)

[13] 陈兴蜀，张帅，董浩等．基于布尔矩阵和 MapReduce 的 FP-Growth 算法 [J]．华南理工大学学报：自然科学版, 2014, 42 (1): 135-141. (Chen Xingshu, Zhang Shuai, Dong Hao, *et al.* FP-Growth algorithm based on Boolean matrix and MapReduce [J]. Journal of South China University of Technology (Natural Science Edition) , 2014, 42 (1): 135-141.)

[14] 高权，万晓东．基于负载均衡的并行 FP-Growth 算法 [J]．计算机工程, 2019, 45 (3): 32-35, 40. (Gao Quan, Wan Xiaodong. Parallel FP-Growth algorithm based on load balance [J]. Computer Engineering, 2019, 45 (3): 32-35, 40.)

[15] Moens S, Aksehirli E, Goethals B. Frequent itemset mining for big data [C]// Proc of International Conference on Advanced Cloud and Big data, 2013: 111-118.

[16] 章志刚，吉根林，唐梦梦．并行挖掘频繁项目集新算法-MREclat [J]．计算机应用, 2014, 34 (8): 2175-2178. (Zhang Zhigang, Ji Genlin, Tang Mengmeng. MREclat: new algorithm for parallel mining frequent itemsets [J]. Journal of Computer Applications, 2014, 34 (8): 2175-2178.)

[17] 张春，汲磊举．基于 MapReduce 的 Eclat 改进算法研究与应用 [J]．北京交通大学学报, 2016, 40 (3): 1-6. (Zhang Chun, Ji Leiju. Research and application of Eclat improved algorithm based on MapReduce [J]. JOURNAL OF BEIJING JIAOTONG UNIVERSITY, 2016, 40 (3): 1-6.)

[18] 冯兴杰，潘轩．基于 Spark 的并行 Eclat 算法 [J]．计算机应用研究,

2019, 36 (1): 18-21. (Feng Xingjie, Pan Xuan. Eclat algorithm based on Spark [J]. Application Research of Computers, 2019, 36 (1): 18-21.)

[19] Liao Jinggui, Zhao Yuelong, Long Saiqin. MRPrePost: a parallel algorithm adapted for mining big data [C]// Proc of IEEE Workshop on Electronics, Computer and Applications, 2014: 564-568.

[20] Deng Zhihong, Wang Zhonghui, Jiang Jiajian. A new algorithm for fast mining frequent itemsets using N-list [J]. Science China Information Sciences, 2012, 55 (9): 2008-2030.

[21] Deng Zhihong. DiffNodesets: An efficient structure for fast mining frequent itemset [J]. Applied Soft Computing, 2016, 41: 214-223.

[22] 尹远，张昌，文凯等. 基于 DiffNodeset 结构的最大频繁项集挖掘算法 [J]. 计算机应用，2018, 38 (12): 3438-3443. (Yin Yuan, Zhang Chang, Wen Kai, *et al.* Maximal frequent itemset mining algorithm based on DiffNodeset structure [J]. Journal of Computer Applications, 2018, 38 (12): 3438-3443.)

[23] http://www. philippe-fournier-viger. com/spmf/index. php?link=datasets. php

[24] 程阳，章韵. 基于 MapReduce-HBase 的 Apriori 算法的改进与研究 [J]. 南京邮电大学学报：自然科学版，2018, 38 (5): 95-103. (Cheng Yang, Zhang Yun. Improvement and research on Apriori algorithm based on MapReduce-HBase [J]. Journal of Nanjing University of Posts and Telecommunications (Natural Science Edition) , 2018, 38 (5): 95-103.)